JOURNAL OF COMPUTER AND SYSTEM SCIENCES 37, 63–78 (1988)

Fast String Matching with k Differences*

GAD M. LANDAU[†]

Department of Computer Science, School of Mathematical Sciences, Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv 69978, Israel

AND

UZI VISHKIN[‡]

Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, New York 10012

Received November 10, 1986; revised February 2, 1988

Consider the string matching problem where differences between characters of the pattern and characters of the text are allowed. Each difference is due to either a mismatch between a character of the text and a character of the pattern or a superfluous character in the text or a superfluous character in the pattern. Given a text of length n, a pattern of length m, and an integer k, we present an algorithm for finding all occurrences of the pattern in the text, each with at most k differences. It runs in $O(m+nk^2)$ time for an alphabet whose size is fixed. For general input the algorithm requires $O(m \log m+nk^2)$ time. In both cases the space requirement is $O(m)$. © 1988 Academic Press, Inc.

1. INTRODUCTION

In the problem of pattern matching in strings (e.g., as discussed in Knuth, Morris, and Pratt [KMP-77]) one is interested in finding all occurrences of a pattern in a text. We consider the problem of *string matching with k differences* (the *k-differences* problem for short), in which the input consists of two strings: a pattern of length m and a text of length n $(n \geqslant m)$ and an integer $k \geqslant 0$, and the output is all occurrences of the pattern in the text with at most k differences.

* The research of these authors was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy under Contract Number DE-AC02-76ER03077.

[†] Present address: Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012.

[‡] Present address: Department of Computer Science, School of Mathematical Sciences, Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv 69978, Israel. The research of this author was supported by NSF Grant NSF-CCR-8615337, ONR Grant N00014-85-K-0046, and Foundation for Research in Electronics, Computers and Communication, Administered by the Israeli Academy of Sciences and Humanities.

We distinguish three types of differences.

(a) A character of the pattern corresponds to a different character of the text. In this case we say that there is a *mismatch* between the two characters.

(b) A character of the pattern corresponds to "no character" in the text (an *insertion*).

(c) A character of the text corresponds to "no character" in the pattern (a *deletion*).

These are illustrated in the following example.

EXAMPLE. Let the text be *abcdefghi*, the pattern *bxdyegh*, and $k = 3$. There is an occurrence with three differences that starts at the second location of the text, given by the following correspondence between *bcdefgh* and *bxdyegh*.

1. b (of the text) corresponds to b (of the pattern),
2. c to x,
3. d to d,
4. nothing to y,
5. e to e,
6. f to nothing,
7. g to g,
8. h to h.

The correspondence can be illustrated as follows,

$$\begin{array}{cccccccccccc} & | & b & x & d & y & e & & g & h & | & \\ a & | & b & c & d & & e & f & g & h & | & i. \end{array}$$

This correspondence has three differences, in positions 2, 4, and 6.

The k-differences problem should be distinguished from the easier problem of *string matching with k mismatches* (the *k-mismatches* problem for short). The k-mismatches problem is to find all occurrences of the pattern in the text with at most k differences of type (a), where differences of type (b) or (c) are not allowed. Ivanov [I-85] gave an algorithm for this problem, whose running time is linear in m and n but which grows very rapidly as function of k. In [LV-86b] we gave an $O(k(m \log m + n))$ algorithm for the k-mismatches problem.

The k-differences problem arises when one needs to analyze situations where the data is not completely reliable. Consider a situation where the strings that are the input for the problem contain errors and one still needs to find all possible occurrences of the pattern in the error-free version of the text. The errors may include a character being replaced by another character, a character being omitted, or a superfluous character being inserted. If one knows a bound on the number of errors, this reduces this problem to the k-differences problem. Applications of the k-differences problem in molecular biology are discussed in [LVN-86]. Sankoff and Kruskal [SK-83] give a comprehensive review of applications of the k-differences problem.

We give two versions of a new algorithm for the k-differences problem, designed for random access machines (RAM) [AHU-74]. The first version, which has a simpler implementation, runs in $O(m^2+nk^2)$ time and requires $O(m^2)$ space. The second version needs $O(m \log m+nk^2)$ time and $O(m)$ space. One can reduce this time bound to a bound linear in m for a fixed alphabet size.

Both these algorithms consist of a pattern-analysis and a text-analysis. The pattern-analysis takes as input the pattern A and computes a matrix MAX-LENGTH(i, j) for $0 \leqslant i,\ j \leqslant m-1$, where MAX-LENGTH$(i, j)$ denotes the maximum length of a substring starting at the $i+1$th position of the pattern A that agrees with a substring starting from the $j+1$th position of A. The text-analysis is given as input the pattern A, text T, and matrix MAX-LENGTH and computes all occurrences in the text having $\leqslant k$ differences with the pattern A. In both versions the text-analysis takes $O(nk^2)$ time. The pattern-analysis in the first version of the algorithm takes $O(m^2)$ time. In the second version it takes $O(m)$ time when the size of the alphabet is fixed. If the alphabet of the pattern contains x ($x \leqslant m$) letters then it is easy to adapt pattern-analysis to run in time $O(m \log x)$.

Despite its slower asymptotic running time, the first version has some advantages compared to the second version. It is simpler and easier to program. In addition, it is useful whenever the time for the text-analysis dominates the computation time. There are a few realistic possibilities where this happens:

1. The pattern is known in advance and we have plenty of time to analyze it.
2. m is sufficiently small with respect to n. Specifically, $m=o(k\sqrt{n})$.
3. The same pattern has to be matched with different texts and m is sufficiently small with respect to the sum of the lengths of these texts.

The case $k=0$ of the k-differences problem is the extensively studied string matching problem. There are a few notable algorithms for the string matching problem: linear time serial algorithms ([BM-77, GS-83, KMP-77, KR-87] (a randomized algorithm) and [V-85]) and parallel algorithms ([G-84, V-85]). None of these algorithms can cope with the k-differences problem.

There has been some previous work on the k-differences problem. A simple $O(mn)$-time dynamic programming algorithm for the k-differences problem was given independently in nine different papers; these are discussed and referenced in Sankoff and Kruskal [SK-83]. Note that our algorithms are asymptotically faster than the dynamic programming algorithm only if $k^2=o(m)$. For the k-differences problem, Ukkonen [U-85] presents an interesting algorithm whose pattern-analysis takes exponential time. The algorithm runs in time $O(m|\Sigma|\, G+n)$ and requires $O((|\Sigma|+m)G)$ space, where $|\Sigma|$ is the size of the alphabet and $G=\min(3^m, 2^k|\Sigma|^k m^{k+1})$. Preprocessing of the pattern takes $O(m|\Sigma|\, G)$ time. Then analysis of the text takes $O(n)$ time. However, the space and time requirements for preprocessing the pattern makes this algorithm impractical, in general. (For comparison, the space requirement of our first version is $O(m^2)$ and that of the second is $O(m)$.)

Our algorithm uses some of the ideas of Ukkonen [U-83] together with the general framework of Knuth, Morris, and Pratt [KMP-77].

This paper contains the main result of our conference paper [LV-85]. That paper also contains an exceedingly simple algorithm for string matching with one difference, which is not included here. In a later paper [LV-86a] we developed a new parallel algorithm for solving the k-differences problem that also leads to a new sequential algorithm. The sequential algorithm of [LV-86a] runs in $O(m+nk)$ time for an alphabet whose size is fixed and requires $O(m \log m + n(\log m + k))$ time for input with unbounded alphabet size. In both cases the space requirement is $O(m+n)$. For general input the algorithm given here is not slower (in order of magnitude) than the later algorithm [LV-86a] when k^2 is $O(\log m)$ and is even asymptotically faster when $k^2 = o(\log m)$. For an alphabet whose size is fixed the latter algorithm is asymptotically faster than the algorithms given here. The algorithm of this paper is interesting for two main reasons. (1) For some cases it is still asymptotically faster then the later algorithm, as specified above. (2) It represents a different algorithmic approach for the k-differences problem, where at each time we search for a match starting at only one location of the text.

Section 2 presents the text-analysis part of the algorithm for the k-differences problem. Section 3 presents the pattern-analysis part. Both Sections 2 and 3 discuss the first version of the algorithm only. The second version is given in Section 4.

2. Analysis of the Text

The *input to the text-analysis* consists of the following:

(a) The pattern: An array $A = a_1, ..., a_m$.

(b) The text: An array $T = t_1, ..., t_n$.

(c) An integer $k \geqslant 0$.

(d) The output of the pattern-analysis: The two-dimensional array MAX-LENGTH $[0, ..., m-1; 0, ..., m-1]$. MAX-LENGTH $(i, j) = f$ means that $a_{i+1}, ..., a_{i+f} = a_{j+1}, ..., a_{j+f}$, and $a_{i+f+1} \neq a_{j+f+1}$. In words, if one places the suffix of the pattern starting at a_{i+1} over the suffix of the pattern starting at a_{j+1}, then MAX-LENGTH (i, j) is the longest exact match of prefixes of these two suffixes.

Output of the text-analysis: All occurrences with $\leqslant k$ differences of the pattern in the text. We start with a very high-level description of the text-analysis algorithm. The algorithm consists of $n-m+k$ iterations. At iterations i a procedure named CHECK checks whether an occurrence of the pattern with $\leqslant k$ differences starts at position $i+1$ of the text.

The Text-Analysis Algorithm

```
for $i := 0$ to $n - m + k$ do
begin
  Apply procedure CHECK
end
```

The rest of this section is devoted to three realizations of procedure CHECK. We first give two easier realizations, which are based on known ideas, and then describe our new technique. The first realization, CHECK – 1, given in Subsection 2.1, is based on a simple dynamic programming algorithm [SK-83]. The second realization, CHECK – 2, given in Subsection 2.2, is based on the work of Ukkonen [U-83]. It follows the same dynamic programming computation in a slightly different way which is also used in our new realization. These two realizations do not use the pattern-analysis precomputation which is item (d) of the input to the text-analysis. Subsection 2.3 gives the new realization CHECK – 3.

2.1. *Dynamic Programming Realization of* CHECK – 1

At each iteration i for $0 \leqslant i \leqslant n - m + k$ we construct a $m + k + 1$ by $m + 1$ matrix $D^{(i)}$ (in short D) whose entries $D^{(i)}_{l,j}$ (in short $D_{l,j}$) count the minimum number of differences between the strings $a_1, ..., a_l$ and $t_{i+1}, ..., t_{i+j}$. If $D_{m,j} \leqslant k$, for at least one j, $m - k \leqslant j \leqslant m + k$, then we conclude that there is an occurrence with at most k differences starting at t_{i+1}.

Procedure CHECK – 1

The following algorithm computes the matrix D

```
Initialization    $D_{0,0} := 0$
                  for all $j$, $1 \leqslant j \leqslant m + k$, $D_{0,j} := j$
                  for all $l$, $1 \leqslant l \leqslant m$, $D_{l,0} := l$,
for $l := 1$ to $m$ do
  for $j := 1$ to $m + k$ do
    $D_{l,j} := \min(D_{l-1,j} + 1; D_{l,j-1} + 1; D_{l-1,j-1}$ if $a_l = t_{i+j}$ or $D_{l-1,j-1} + 1$
    otherwise)
    ($D_{l,j}$ is the minimum of three numbers. These numbers are obtained from the
    predecessors of $D_{l,j}$ on its column, row, and diagonal, respectively.)
  od
od
for $j := m - k$ to $m + k$ do
  if $D_{m,j} \leqslant k$
  then print* An occurrence with $\leqslant k$ differences of the pattern starts at $t_{i+1}$*
od
```

Complexity analysis of CHECK – 1. Each application of CHECK – 1 takes

$O(m^2)$ time. (This is since it does not make sense to have $k > m$.) Therefore, the running time of the text-analysis algorithm is $O(nm^2)$. The algorithm as stated uses $O(m^2)$ space, since the matrix D is computed separately at each iteration. In fact the algorithm is easily modified to use $O(m)$ space, since during each iteration only two rows $D_{l,j}$ and $D_{l-1,j}$ for $0 \leqslant j \leqslant m+k$ need to be stored at any time.

2.2. *Modified Dynamic Programming Realization* CHECK – 2[U-83]

Ukkonen [U-83] found a more efficient way to compute the matrix D. Let the iteration i be fixed. By *diagonal d* of the matrix D we mean all entries $D_{l,j}$ such that $d = j - l$. (Here $-k \leqslant d \leqslant k$.)

LEMMA 1 [U-83]. *For every l, j, $D_{l,j} - D_{l-1,j-1}$ is either zero or one.*

This lemma simplifies the recursion in the inner loop of CHECK – 1 to

$$D_{l,j} = \begin{cases} D_{l-1,j-1}, & \text{if } a_l = t_{i+j} \\ \text{Min}(D_{l-1,j} + 1, D_{l,j-1} + 1, D_{l-1,j-1} + 1, & \text{if } a_l \neq t_{i+j}. \end{cases}$$

In addition it can be used to store the information of the matrix D in a more compact way. For a number of differences e and a diagonal d, let $L_{d,e}$ denote the largest row l such that $D_{l,j} = e$ and $D_{l,j}$ is on diagonal d. Note that this implies that there are e differences between $a_1, ..., a_{L_{d,e}}$ and $t_{i+1}, ..., t_{i+L_{d,e}+d}$, and $a_{L_{d,e}+1} \neq t_{i+L_{d,e}+d+1}$.

To determine whether a match with $\leqslant k$ differences occurs between the pattern and the text starting at t_{i+1}, one needs only to compute the values of $L_{d,e}$, where e and d satisfy $e \leqslant k$ and $|d| \leqslant e$. To see this, we note that $e \leqslant k$ is obvious. The initial values $D_{0,j}$ and $D_{l,0}$ together with Lemma 1 guarantee that all $D_{l,j}$ on diagonal d are $\geqslant |d|$ and therefore given a number of differences e we need only values of $L_{d,e}$ where $|d| \leqslant e$.

If one of the $L_{d,e}$ ($|d| \leqslant e \leqslant k$) equals m, we conclude that an occurrence of the pattern with at most k differences starts at t_{i+1}. If for every diagonal d, $|d| \leqslant k$, $L_{d,k}$ (is defined and) is less than m then we conclude that any correspondence of the pattern and the text starting at t_{i+1} must have at least $k+1$ differences.

Now we derive a recursion for the quantities $L_{d,e}$ in terms of the neighboring diagonal quantities $L_{d-1,e-1}$, $L_{d+1,e-1}$. We note that $D_{l,j}$ on diagonal d is assigned a value $\geqslant e$ only if all the following occur:

(a) either $D_{l-1,j-1} \geqslant e$ or $D_{l-1,j-1} \geqslant e-1$ and $a_l \neq t_{i+j}$,

(b) the element $D_{l,j-1} \geqslant e-1$ (this element is on the diagonal "below" d),

(c) the element $D_{l-1,j} \geqslant e-1$ (this element is on the diagonal "above" d).

It is now easy to see that if

$$\text{row} = \max[(L_{d,e-1}+1), (L_{d-1,e-1}), (L_{d+1,e-1}+1)],$$
$$h^{**} = \max(h^{*}: a_{\text{row}+h} = t_{i+d+\text{row}+h} \text{ for } 1 \leqslant h \leqslant h^{*}) \text{ where } h^{**}=0 \text{ by default},$$
$$L_{d,e} = \min(\text{row}+h^{**}, m).$$

These observations yield the following procedure CHECK − 2 for computing the $L_{d,e}$'s ($|d| \leqslant e \leqslant k$) which proceeds by incrementing e.

Procedure CHECK − 2

```
[1] Initialization
  for $d := -(k+1)$ to $(k+1)$ do
    $L_{d,|d|-2} := -\infty$
    if $d<0$
      then $L_{d,|d|-1} := |d|-1$
      else $L_{d,|d|-1} := -1$
  od
[2] for $e := 0$ to $k$ do
  for $d := -e$ to $e$ do
    [3] row $:= \max[(L_{d,e-1}+1), (L_{d-1,e-1}), (L_{d+1,e-1}+1)]$
        row $:= \min(\text{row}, m)$
    (Goal of instruction [4]: Read just enought additional text to force $e+1$ differences on diagonal $d$, or reach the end of the pattern or the text, whichever comes first.)
    [4] while row $< m$ and $i+\text{row}+d<n$ and $a_{\text{row}+1} = t_{i+\text{row}+1+d}$ do
              row $:=$ row $+ 1$
      od
    [5] $L_{d,e} :=$ row
    [6] if $L_{d,e} = m$
    then print* An occurrence with $\leqslant k$ differences of the pattern starts at $t_{i+1}$*
  od
od
```

Remark. The values in the Initialization step are fictitious, but lead to proper initialization of the $L_{d,e}$ values on the boundaries of the matrix.

Correctness of procedure CHECK − 2. One need only check that the initialization values are correct, since the recursion to compute $L_{d,e}$ is essentially that given above. Let $e=0$ and $d=0$. Consider the computation of $L_{0,0}$. Instruction 3 starts by initializing row to 0. Instructions 4 and 5 find that $a_1, ..., a_{L_{0,0}}$ is equal to $t_{i+1}, ..., t_{i+L_{0,0}}$ and $a_{L_{0,0}+1} \neq t_{i+L_{0,0}+1}$. Therefore $L_{0,0}$ gets its correct value. To finish the base of the induction we note that for $d \neq 0$, $L_{d,|d|-1}$ and $L_{d,|d|-2}$ get correct values in the Initialization.

Complexity analysis of CHECK − 2. For each iteration i of the text-analysis, CHECK − 2 computes $L_{d,e}$ for the $2k+1$ diagonal values $-k \leqslant d \leqslant k$. For each diagonal d, the variable row can take at most m different values (even as e varies in the course of the algorithm). Therefore, each application of CHECK − 2 takes $O(mk)$ steps, and the whole text-analysis algorithm runs in time $O(nmk)$.

2.3. *The New Procedure* CHECK − 3

The main idea of our speedup is to decrease the time spent in instruction [4] of CHECK − 2. That step detects a longest exact match of a substring of the pattern starting at a given point with a substring of the text starting at a given point. We will show that one can sometimes skip or shorten this step by saving information from earlier stages of CHECK − 2 concerning such matches. In order to effectively use this information, we must store a table MAX-LENGTH of maximum matches between substrings of the pattern starting at different positions. Such an idea appeared in Knuth, Morris, and Pratt [KMP-77].

Now we explain the idea in more detail. Consider iteration i. The procedure CHECK − 3 consists of two stages. All the novel ideas are in the first stage. By the end of iteration $i-1$ the following has been computed. For each location $x = 1, 2, ..., i$ in the text, we have computed the longest prefix of $t_x, t_{x+1}, ...$ that can be matched with a prefix of the pattern in at most k differences. Let $t_x, t_{x+1}, ..., t_{j(x)}$ be this prefix of $t_x, t_{x+1}, ...$. Let j be the maximum over $j(1), j(2), ..., j(i)$. That is, j is the rightmost location of the text which has been reached in a previous iteration. Denote by r the earliest iteration in which location j was reached. (Observe that $r = r(i)$ and $j = j(r)$.) The first stage analyzes the number of differences between $t_{i+1}, t_{i+2}, ..., t_j$ and a prefix of the pattern. If the first stage exposes $k+1$ differences, we proceed to the next iteration. Otherwise, in the second stage we search beyond location j in the text in a similar fashion to the procedure CHECK − 2.

We explain how the relevant information from the previous iterations of the text-analysis is maintained. We defined t_j to be the rightmost symbol in the text that was reached at an iteration prior to i and r to be the earliest iteration at which we reached t_j, $0 \leqslant r < i$. At iteration r we found a correspondence between some prefix of the pattern and $t_{r+1}, ..., t_j$ with at most $k+1$ differences. (Specifically, if an occurrence was found then the number of differences is at most k. If not, then in addition to finding that there is no occurrence, iteration r also finds a correspondence with exactly $k+1$ differences). This correspondence also gives a correspondence with at most $k+1$ differences between some suffix of this prefix of the pattern and $t_{i+1}, ..., t_j$. (We call this suffix of prefix of the pattern the *subpattern.*) There are at least $j-i-k-1$ symbols of $t_{i+1}, ..., t_j$ that have a match in the subpattern. All the symbols of $t_{i+1}, ..., t_j$ that have contiguous matches in this correspondence form at most $k+1$ (contiguous) substrings. (That is, before the first difference, between the first and second differences, and finally between the kth and the $k+1$st difference. See also the example below.) For each such substring we know its corresponding substring in the pattern. Suppose a substring of the text $t_{p+1}, ..., t_{p+f}$ matches a substring of the pattern $a_{c+1}, ..., a_{c+f}$ and $t_{p+f+1} \neq a_{c+f+1}$, we denote this by the *triple* (p, c, f). There ar at most $k+1$ symbols in $t_{i+1}, ..., t_j$ which do not have matching symbol in the subpattern. We denote each such symbol t_{h+1} by the triple $(h, 0, 0)$. The correspondence between $t_{i+1}, ..., t_j$ and the subpattern can be described by a sequence of such triples. In such sequence there are at most $2k+2$ triples. We denote the sequence by $S_{i,j}$.

EXAMPLE. Let the text $t_{17}, ..., t_{30}$ be *abaaacddacdcac*, and the pattern prefix $a_1, ..., a_{13}$ be *aaaaeddcdcbab*, and $i=20$ and the number of differences $k=4$. Suppose $r=16$ and $j=30$. The correspondence

	1		2	3	4	5	6	7		8	9	10	11	12	13
pattern =	*a*		*a*	*a*	*a*	*e*	*d*	*d*		*c*	*d*	*c*	*b*	*a*	*b*
text =	*a*	*b*	*a*	*a*	*a*	*c*	*d*	*d*	*a*	*c*	*d*	*c*		*a*	*c*
	17	18	19	20	21	22	23	24	25	26	27	28		29	30

gives five differences. It can be easily checked that a correspondence with fewer differences is impossible. The *subpattern* is $a_4, ..., a_{13}$, and $S_{20,30}$ is $\{(20, 3, 1)$, $(21, 0, 0)$, $(22, 5, 2)$, $(24, 0, 0)$, $(25, 7, 3)$, $(28, 11, 1)$, and $(29, 0, 0)\}$. (Note that the positions of $t_{17}, ..., t_{30}$ that formed (in this case exactly) $k+1=5$ contiguous substrings of matches between the text and the pattern were [17], [19, 20, 21], [23, 24], [26, 27, 28], and [29].)

Now we describe CHECK − 3 verbally. The main difference between CHECK − 2 and CHECK − 3 is in instruction 4. CHECK − 3 has a new instruction 4.new added which represents the first stage of CHECK − 3, which attempts to skip the second stage which is the default instruction 4.old which duplicates instruction 4 of CHECK − 2. The **while** loop of instruction 4.new looks for the longest exact match of a text substring starting $t_{i+\text{row}+d+1}$ with a pattern substring starting from $a_{\text{row}+1}$, provided this text substring does not extend past t_j, using information from previous iterations. We explain how it looks for the maximum w such that $a_{\text{row}+1}, ..., a_{\text{row}+w}$ equals $t_{i+\text{row}+d+1}, ..., t_{i+\text{row}+d+w}$. Suppose that according to $S_{i,j}$ the substring $t_{i+\text{row}+d+1}, ..., t_{i+\text{row}+d+f}$ matches $a_{c+1}, ..., a_{c+f}$ for some index c of the pattern and some integer $f \geqslant 0$. We can find c and f using the fact that for each t_{h+1} $(i \leqslant h < j)$ there exists a triple (p_1, c_1, f_1) in S_{ij} such that $p_1 \leqslant h \leqslant p_1 + f_1$. (In this case we say that (p_1, c_1, f_1) *covers* t_{h+1}). For the computation of w we start with $w=0$ and consider the following cases:

Case a. $f=0$.

Case a1. $t_{i+\text{row}+d+1} \neq a_{\text{row}+1}$. Hence, we finished the computation of w (it is assigned its present value).

Case a2. $t_{i+\text{row}+d+1} = a_{\text{row}+1}$. Therefore, we assign row := row + 1, add 1 to w, and start the case analysis over.

Case b. $f \geqslant 1$. MAX-LENGTH(c, row) gives the maximal number g such that $a_{c+1}, ..., a_{c+g}$ equals $a_{\text{row}+1}, ..., a_{\text{row}+g}$.

Case b1. $f \neq g$. It is easy to see that here $t_{i+\text{row}+d+1}, \ldots, t_{i+\text{row}+d+\min(f,g)} = a_{\text{row}+1}, \ldots, a_{\text{row}+\min(f,g)}$ and $t_{i+\text{row}+d+\min(f,g)+1} \neq a_{\text{row}+\min(f,g)+1}$. Therefore, we finish the computation of w by assigning $w := w + \min(f, g)$.

Case b2. $f = g$. This implies $t_{i+\text{row}+d+1}, \ldots, t_{i+\text{row}+d+f} = a_{\text{row}+1}, \ldots, a_{\text{row}+f}$ but does not reveal whether $t_{i+\text{row}+d+f+1}$ equals $a_{\text{row}+f+1}$ or not. Therefore, we assign row := row + f,'add f to w, and again apply the present case analysis accumulating this "jump" over f symbols into w.

If $i + \text{row} + d + 1 > j$ occurs, we halt and proceed on to stage 4.old.

A high-level version of the resulting Procedure CHECK – 3 is as follows.

```
*Procedure* CHECK – 3
**begin**
  [1] *Initialization*
    **for** $d := -(k+1)$ **to** $(k+1)$ **do**
      $L_{d,|d|-2} := -\infty$
      **if** $d < 0$
        **then** $L_{d,|d|-1} := |d| - 1$
        **else** $L_{d,|d|-1} := -1$
    **od**
  [2] **for** $e := 0$ **to** $k$ **do**
    **for** $d := -e$ **to** $e$ **do**
      [3] row := max[($L_{d,e-1}+1$), ($L_{d-1,e-1}$), ($L_{d+1,e-1}+1$)]
      [4.new] **while** $i + \text{row} + d + 1 \leqslant j$ **do**
        ($S_{i,j}$ gives the details of an approximate match between the subpattern
        and the text from position $i$ through position $j$, which is the end of the text
        prefix that has been read so far. *Goal of instruction* [4.new]: Extract just
        enough information from $S_{i,j}$ to force $e+1$ differences on diagonal $d$, or
        reach position $j$.)
        [4.new.1] take from $S_{i,j}$ the triple that "covers" $t_{i+\text{row}+d+1}$. Derive from
        this triple the indices $c, f$ such that $t_{i+\text{row}+d+1}, \ldots, t_{i+\text{row}+d+f} =$
        $a_{c+1}, \ldots, a_{c+f}$ ($t_{i+\text{row}+d+f+1} \neq a_{c+f+1}$)
        [4.new.2] **if** $f = 0$
        **then** (*case a*)
          [4.new.3] **if** $t_{i+\text{row}+d+1} \neq a_{\text{row}+1}$
          **then** (*case a1*)
            **go to** 5
          **else** (*case a2*)
            row := row + 1
        **else** (*case b*)
          [4.new.4] **if** $f \neq$ MAX-LENGTH (c, row)
          **then** (*case b1*)
            row := row + min($f$, MAX-LENGTH($c$, row))
            **go to** 5
            else (*case b2*)
              row := row + $f$
        od
        (*Goal of instruction* [4.old]: Read just enought additional text to force $e+1$
        differences on diagonal $d$, or reach the end of the pattern or the text,
        whichever comes first.)
        [4.old] while row $< m$ and $i +$ row $+ d < n$ and $a_{\text{row}+1} = t_{i+\text{row}+1+d}$ do
          row := row + 1
        od
        [5] $L_{d,e}$ := row
        [6] if $L_{d,e} = m$
        then print* An occurence with $\leqslant k$ differences of the pattern starts at $t_{i+1}$*
      od
    od
    [7] If new symbols of the text were reached, reconstruct a new $S_{i,j}$.
end
```

To obtain a complete algorithm, we must explain how to do instructions [4.new.1] and [7] which were not fully specified in the description above.

Instruction 4.new.1. We find the triple of $S_{i,j}$ that covers $t_{i+\text{row}+d+1}$, each time this instruction is performed as follows. For each diagonal d the values of $i+\text{row}+d+1$ increase monotonically each time instruction 4.new.1 is performed. Therefore, looking up for the triple that covers $t_{i+\text{row}+d+1}$ involves monotonic advancement through the triples of $S_{i,j}$.

Instruction 7. At the end of each iteration i if at least one new symbol of the text was reached we have to create a new sequence of triples instead of $S_{i,j}$. We do it as follows. If $t_{\bar{j}}$ is the rightmost symbol of the text that was reached in such an iteration then denote the new sequence $S_{i,\bar{j}}$. In order to compute $S_{i,\bar{j}}$ we find for each $L_{d,e}$, in the course of the computation, a sequence $T(d, e)$ of such triples that "realizes" $l = L_{d,e}$ in the sense that it describes a match of $t_{i+1}, t_{i+2}, ..., t_{i+l+d}$ with $a_1, ..., a_l$ that has exactly e differences. At any given time we will maintain at most $2k+1$ such lists (for the current and previous value of e) and they will occupy $O(k^2)$ space. At the beginning of each iteration i, each $T(d, e)$ is empty. We again use the fact that initially (at instruction 3) row is the maximum among $L_{d-1,e-1}$, $L_{d,e-1}+1$, and $L_{d+1,e-1}+1$ and finally (at instruction 5) is $L_{d,e}$. Assume that *we know the sequences of the predecessors of* $L_{d,e}$ (namely, $T(d-1, e-1)$, $T(d, e-1)$, and $T(d+1, e-1)$). We get the sequence of $L_{d,e}(T(d, e))$ by adding triples to the end of the sequence of the predecessor that gives the maximum in initializing row. Let r_1 be the initial value of row. If r_1 got its value from $L_{d-1,e-1}$ (or $L_{d,e-1}$) then we first add the triple $(i+r_1+d-1, 0, 0)$. (Meaning that for t_{i+r_1+d}, there is no corresponding symbol in the pattern.) Following instruction 5, if $L_{d,e} > r_1$, we next add the triple $(i+r_1+d, r_1, L_{d,e}-r_1)$. This is done regardless of whether the source of r_1 was $L_{d-1,e-1}$ or $L_{d,e-1}$ or $L_{d+1,e-1}$. (This triple describes the match between substrings of the pattern and the text that was found during the computation of $L_{d,e}$ given $L_{d-1,e-1}$, $L_{d,e-1}$, and $L_{d+1,e-1}$.) At the end of iteration i we check which of the $2k+1$ sequences reached the rightmost symbol of the text. If the index of this symbol is greater than j ($L_{d,e}+d+i>j$), then we take its sequence to be the new $S_{i,j}$ and $L_{d,e}+d+i$ to be the new j.

Correctness of procedure CHECK $-$ 3. This follows by induction on the iteration number. We show that at each iteration i we find whether there is an occurrence of the pattern in the text with at most k differences. The first iteration is the same as the first iteration in CHECK $-$ 2. Also instruction 7 of the first iteration computes $S_{0,j}$ properly. Suppose each iteration $x=1, 2, ..., i-1$ found whether there is an occurrence with at most k differences starting at t_{x+1} and if necessary computed a new $S_{x,j}$. The inductive hypothesis guarantees that iteration i has the required input. The above verbal description explains why iteration i will find whether an occurrence of the pattern with at most k differences starts at t_{i+1}. Instruction 7 computes $S_{i,j}$ if necessary.

Complexity analysis of CHECK $-$ 3. We claim that each iteration of Procedure CHECK $-$ 3 runs in $O(k^2)$ steps using $O(k^2)$ storage, so that the whole text-analysis requires $O(nk^2)$ steps.

The instruction [4.old] (where row is increased by one at a time without using $S_{i,j}$ and MAX-LENGTH) is employed each time we move to a new symbol of the text. We maintain $O(k)$ diagonals and may need to compare the new symbol for each of them, for $O(k)$ time in instruction [4.old] in CHECK $-$ 3.

In order to evaluate the number of steps that are required instruction [4.new], we again use the fact that $O(k)$ diagonals are computed, in each iteration. The sequence $S_{i,j}$ has at most $2k+2$ triples. We can charge each operation performed on any one of the diagonals to either a difference being discovered (there are $\leqslant k+1$ such differences), or to a triple of $S_{i,j}$ being examined (there are $\leqslant 2k+2$ triples). This amounts to $O(k)$ operations per diagonal and $O(k^2)$ per iteration.

Finally instruction [7] takes $O(k^2)$ steps in all, in the process of computing all the $T(d, e)$, because the new $T(d, e)$'s are created by adding triples to old $T(d, e)$'s, and only $O(k^2)$ triples are added in all. This proves the claim, completing the complexity analysis.

3. String Matching with k Differences—Pattern-Analysis (First Version)

The input to the pattern-analysis is the pattern, which is given as an array $A=a_1, ..., a_m$. The output of the pattern-analysis is a two-dimensional array MAX-LENGTH[0, ..., $m-1$; 0, ..., $m-1$]. MAX-LENGTH$(i, j)=f$ means that $a_{i+1}, ..., a_{i+f}=a_{j+1}, ..., a_{j+f}$, and $a_{i+f+1}\neq a_{j+f+1}$. In words, consider laying the suffix of the pattern starting at a_{i+1} over the suffix of the pattern starting at a_{j+1}. MAX-LENGTH(i, j) is the length of the longest exact match of prefixes of these two suffixes.

The following procedure MAX-LENGTH computes this array by a simple recursion. Define the pair (i, j), $0 \leqslant i, j \leqslant m-1$, to be on *diagonal* d if $j-i=d$ where the possible values of d are $-(m-1) \leqslant d \leqslant m-1$. We compute each diagonal d of MAX-LENGTH separately. Below, we give the algorithm for any diagonal d, $0 \leqslant d \leqslant m-1$. The algorithm for diagonals d where $-(m-1) \leqslant d < 0$ is similar. Note, however, that a trivial change in the initialization step is required.

```
Procedure MAX-LENGTH
begin
  Initialization
    for $d := 0$ to $m-1$ do
      MAX-LENGTH$(m-1-d, m-1) := 1$ if $a_{m-d} = a_m$
      MAX-LENGTH$(m-1-d, m-1) := 0$ otherwise
    od
  for $d := 0$ to $m-1$ do
    for $i := m-2-d$ to 0 do
      if $a_{i+1} = a_{i+d+1}$
        then MAX-LENGTH$(i, i+d) := 1 +$ MAX-LENGTH$(i+1, i+d+1)$
        else MAX-LENGTH$(i, i+d) := 0$
    od
  od
end
```

Complexity analysis. The number of operations performed by the algorithm for each diagonal is proportional to the number of pairs on the diagonal. Therefore, the total number of operations performed by the algorithms for all diagonals is proportional to the total number of pairs, which is $O(m^2)$.

4. String Matching with k Differences (Second Version)

We obtain an improved pattern-analysis procedure by computing a *suffix tree* instead of the table MAX-LENGTH. At the end of this section we describe how the information in the suffix tree can be used directly in the text-analysis algorithm in place of the table MAX-LENGTH.

The input to the suffix tree algorithm is a pattern concatenated with a special symbol $\$$, which does not appear elsewhere in the pattern $(a_1, \ldots, a_m\$)$.

We define the *suffix tree* of the pattern as follows:

(1) All the edges of the tree are directed away from the root. The out-degree of each node of the tree is either zero (if the node is a leaf) or $\geqslant 2$.

(2) Each edge of the tree corresponds to some (contiguous) substring of the pattern $a_i, a_{i+1}, \ldots, a_j$, where $1 \leqslant i \leqslant j \leqslant m$. The substrings of two sibling edges (edges emanating from the same vertex of the tree) cannot have identical (nonempty) prefixes. For each node v of the tree there is a directed path from the

root to v. Concatenating all substrings that correspond to edges along this path yields a string which corresponds to v.

(3) The tree has m leaves, each corresponding to a different suffix of the pattern

Up to isomorphism (of graphs) there is only one suffix tree for a given pattern.

EXAMPLE. Given the pattern *abab$* the suffix tree is as shown in Fig. 1a.

The output of the procedure PATTERN-ANALYSIS will be the suffix tree of the pattern (Fig. 1b), in which each node v of the tree (besides the root) has (1) a pointer to its father (the node on the other side of its single incoming edge in the tree); (2) LENGTH(V)—the number of characters in the string that corresponds to the path from the root to v. (Note that the number of characters in the string that corresponds to the path from the root to a leaf readily implies which suffix corresponds to this leaf.) We compute the suffix tree using the algorithm of Weiner [W-73].

Complexity analysis of procedure PATTERN-ANALYSIS. We refer the reader to Weiner [W-73] for computation of the suffix tree in $O(m)$ time when the size of the alphabet is fixed. If the alphabet of the pattern contains x letters then it is easy to adapt the algorithm of [W-73] and the whole pattern-analysis to run in time $O(m \log x)$. In both cases the space requirement of the pattern-analysis is $O(m)$. (The reader is also referred to Chen and Seiferas [CS-85] for a lucid presentation of the algorithm of [W-73].)

Modified text-analysis procedure CHECK − 4. It remains to describe how the text-analysis procedure CHECK − 3 must be modified to use the output of PATTERN-ANALYSIS. We call the modified procedure CHECK − 4. The array MAX-LENGTH that was used in CHECK − 3 had the following information. MAX-LENGTH$(i, j) = f$ meant that $a_{i+1}, ..., a_{i+f}$ is equal to $a_{j+1}, ..., a_{j+f}$ and

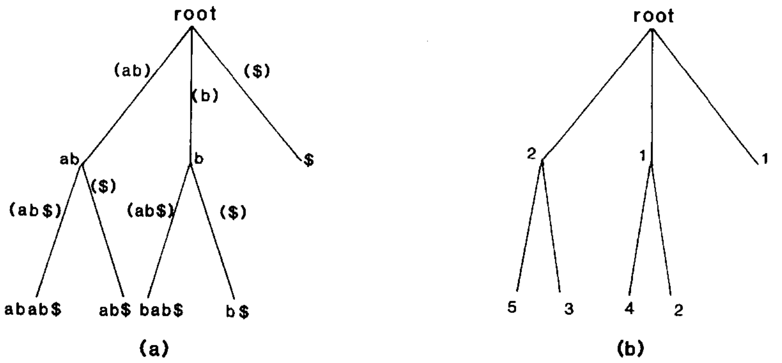

FIG. 1. (a) The suffix tree. (b) The output of the PATTERN-ANALYSIS.

$a_{i+f+1} \neq a_{j+f+1}$. The idea now is to consider each request for MAX-LENGTH(i, j) to be a query that has to be satisfied. So, it remains to show how to satisfy the query MAX-LENGTH(i, j) using the output of procedure PATTERN-ANALYSIS. Let LCA$_{i,j}$ be the lowest common ancestor (in short LCA of the leaves corresponding to the suffixes $a_{i+1}, ..., a_m$ and $a_{j+1}, ..., a_m$ in the suffix tree. The desired MAX-LENGTH(i, j) is simply LENGTH(LCA$_{i,j}$). Thus, the problem of computing the query MAX-LENGTH(i, j) is reduced to finding LCA$_{i,j}$. We use the algorithm of Harel and Tarjan [HT-84] (or the simpler algorithm of Scheiber and Vishkin [SV-88]) for the purpose of computing all the queries that arise throughout the modified text-analysis algorithm CHECK – 4.

Complexity analysis of CHECK – 4. Each query of the form MAX-LENGTH(i, j) requires $O(1)$ time in the first version of the algorithm. Since the analysis of the text took there $O(nk^2)$ time, only $O(nk^2)$ such queries may arise throughout the text-analysis. Using the classification of Harel and Tarjan [HT-84], we are interested in the *static lowest common ancestors* problem, where the tree is static but the queries for the lowest common ancestors of pairs of vertices are given on line. That is, each query must be answered before the next one is known. The suffix tree that is the output of the pattern-analysis has $O(m)$ nodes. The algorithm of [HT-84] proceeds as follows. It preprocesses the suffix tree in $O(m)$ time and $O(m)$ space. Then, given an LCA query it responds in $O(1)$ time. Applying their algorithm clearly yields $O(nk^2)$ time. Beyond MAX-LENGTH, the additional space requirements of CHECK – 3 is $O(k)$. Therefore, we need $O(m+k) = O(m)$ space for the text-analysis of CHECK – 4.

ACKNOWLEDGMENTS

We are grateful to Ed Schonberg for his comment that simplified considerably the pattern-analysis part of the first version of the algorithm. Numerous helpful suggestions by the referees are gratefully acknowledged.

REFERENCES

[AHU-74] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, MA, 1974.

[BM-77] R. S. BOYER AND J. S. MOORE, A fast string searching algorithm, *Comm. ACM* **20** (1977), 762–772.

[CS-85] M. T. CHEN AND J. SEIFERAS, Efficient and elegant subword tree construction, *in* "Combinatorial Algorithms on Words," (A. Apostolico and Z. Galil, Eds.), NATO ASI Series, Series F: Computer and System Sciences, Vol. 12, pp. 97–107, Springer-Verlag, Berlin/New York, 1985.

[G-84] Z. GALIL, Optimal parallel algorithms for string matching, *in* "Proc. 16th ACM Symposium on Theory of Computing 1984," pp. 240–248; also, *Inform. and Control* **67** (1985), 144–157.

[G-85] Z. GALIL, Open problems in stringology, *in* "Combinatorial Algorithms on Words" (A. Apostolico and Z. Galil, Eds.), NATO (ASI) Series, Series F: Computer and System Sciences, Vol. 12, pp. 1–8, Springer-Verlag, Berlin/New York, 1985.

[GS-83] Z. GALIL AND J. I. SEIFERAS, Time-space-optimal string matching, *J. Comput. System Sci.* **26** (1983), 280–294.

[HT-84] D. HAREL AND R. E. TARJAN, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.* **13**, No. 2 (1984), 338–355.

[I-85] A. G. IVANOV, Recognition of an approximate occurrence of words on a Turing machine in real time, *Math. USSR-Izv.* **24**, No. 3 (1985), 479–522.

[KMP-77] D. E. KNUTH, J. H. MORRIS, AND V. R. PRATT, Fast pattern matching in strings, *SIAM J. Comput.* **6** (1977), 323–350.

[KR-87] R. M. KARP AND M. O. RABIN, Efficient randomized pattern-matching algorithms, *IBM J. Res. Develop.* **31**, No. 2 (March 1987), 249–260.

[LV-85] G. M. LANDAU AND U. VISHKIN, Efficient string matching in the presence of errors, *in* "Proc. 26th IEEE FOCS, 1985," pp. 126–136.

[LV-86a] G. M. LANDAU AND U. VISHKIN, Introducing efficient parallelism into approximate string matching, *in* "Proc. 18th ACM Symposium on Theory of Computing, 1986", pp. 220–230.

[LV-86b] G. M. LANDAU AND U. VISHKIN, Efficient string matching with k mismatches, *Theoret. Comput. Sci.* **43** (1986), 239–249.

[LVN-86] G. M. LANDAU, U. VISHKIN, AND R. NUSSINOV, An efficient string matching algorithm with k differences for nucleotide and amino acid sequences, *Nucleic Acid Res.* **14**, No. 1 (1986), 31–46.

[SK-83] D. SANKOFF AND J. B. KRUSKAL (Eds.), "Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, Addison-Wesley, Reading, MA, 1983.

[SV-88] B. SCHIEBER AND U. VISHKIN, Parallel computation of lowest common ancestor in trees, *SIAM J. Comput.* to appear.

[U-83] E. UKKONEN, On approximate string matching, *in* "Proc. Int. Conf. Found. Comp. Theor.," Lecture Notes in Computer Science 158, pp. 487–495, Springer-Verlag, Berlin/New York, 1983.

[U-85] E. UKKONEN, Finding approximate pattern in strings, *J. Algorithms* **6** (1985), 132–137.

[V-85] U. VISHKIN, Optimal parallel pattern matching in strings, *in* "Proc. 12th ICALP," Lecture Notes in Computer Science 194, pp. 497–508, Springer-Verlag, Berlin/New York, 1985; also, *Inform. and Control* **67** (1985), 91–113.

[W-73] P. WEINER, Linear pattern matching algorithm, *in* "Proc. 14th IEEE Symposium on Switching and Automata Theory, 1973," pp. 1–11.